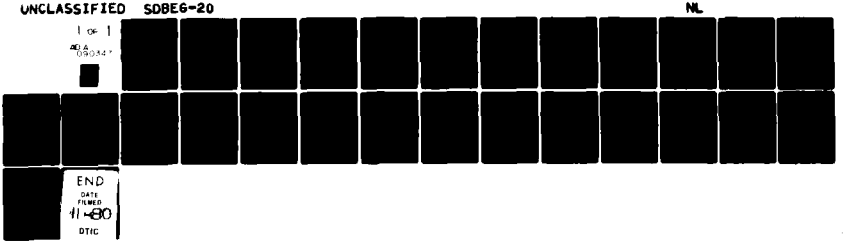
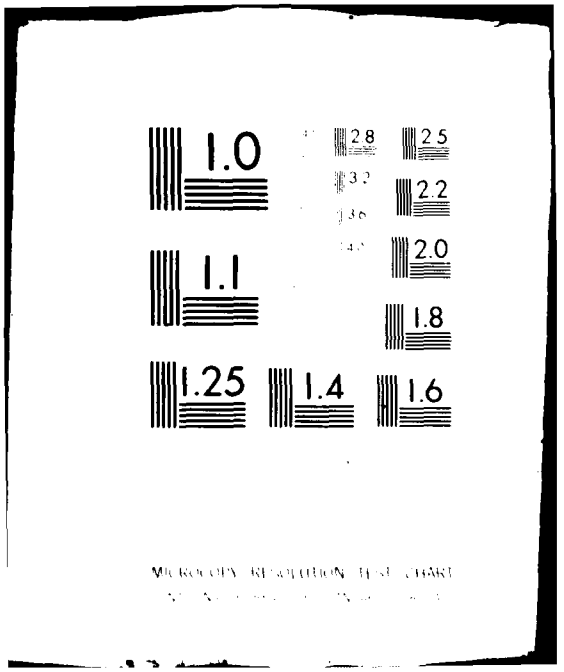
AD-A090 347 TEXAS UNIV AT AUSTIN SOFTWARE AND DATA BASE ENGINEER--ETC F/G 9/2
AN INTEGRATED METHODOLOGY AND TOOLS FOR SOFTWARE DEVELOPMENT.(U)
AUG 80 D CHESTER, R T YEH AFOSR-77-3409

UNCLASSIFIED SDBE6-20 NL

1 OF 1
AD A 090347



END
DATE FILMED
11-80
DTIC


1.0
2.8
2.5
3.2
2.2
3.6
2.0
1.1
1.8
1.25
1.4
1.6
MICROCOPY RESOLUTION TEST CHART

AFOSR-TR- 80-0967

LEVEL II

(9)

AD A090347

DTIC
ELECTE
OCT 15 1980
S
C

SOFTWARE AND DATA BASE ENGINEERING

Department of Computer Science

DDC FILE COPY

Approved for public release;
distribution unlimited.

80 10 9 090

(9)

AN INTEGRATED METHODOLOGY AND TOOLS FOR SOFTWARE DEVELOPMENT

(6) [redacted]RT

Daniel Chester (10)
Raymond T. Yeh

(9) Final Report (14) SDBEC-20

August 1980 (11)

(12) 25

Funds for this research were derived from the Air Force Office of Scientific Research under Grant AFOSR-77-3409 (15)

411 13

ABSTRACT

This research has developed a methodology for the development of software and several tools to help the designers of software systems. The methodology incorporates models of various aspects of a system under opment. They include models of how efficiently the system will m, what function the system will compute, and what alternatives signers have considered and decided against. The methodology was to data base systems, where it works in a way that agrees with experience with those systems. The tools that were developed include f system performance, specification languages, a design tion language, facilities for modelling conceptual entities and ys to control the complexity of processes that are operating el (concurrent programs).

I INTRODUCTION

Although many activities are being carried out in software engineering today, it is far from a mature engineering discipline. Part of the reason is that the academic community is interested in more long term research (such as program correctness, automatic programming, etc.) and industry is busy in developing ad hoc aids to correct its immediate problems. As a result, software engineering as a discipline urgently needs engineering aids for software designers.

In order to provide engineering aids to software designers, it is necessary, in our opinion, to enhance current software design methods based on a step-wise refinement approach to incorporate design evaluation methods which are based on performance, and by developing languages and tools for documenting intermediate design decisions to control system evolution. Furthermore, any such enhanced methodology needs to be verified by application to real system construction. It is our opinion that data base systems (DBS) is a class of systems which can be used for testing the methodology.

II RESEARCH OBJECTIVES

Our research at the University of Texas has been directed for the last three years toward the following engineering aids:

a) A methodology for software development. This is to be a practical adaptation of current ideas like step-wise refinement [Dijkstra 1972]. We want to divide the development of a software system into a sequence of stages in such a way that as many errors as possible can be caught in the early stages of the design process.

b) Hierarchical performance models. We proposed to develop a hierarchy of models; crude models which employ only gross characteristics of subsystems would be used at the top level while more refined models would be used as the design evolves. These models help designers to evaluate the performance of their designs at an earlier stage of design than is done currently. By discovering errors before a fine level of detail has been worked out, designers should save a

significant amount of redesign and reprogramming effort.

c) A specification language for constructing "mock-ups". A common approach used in other engineering disciplines is to build a "mock-up" or "prototype" so that the designer will have an opportunity to investigate a scaled down model before construction takes place. This specification language is intended to provide the software designer with a working model of his design so that he can test whether it behaves as intended before he works out the finer details of the design.

d) A software design documentation language. This language is to be used to express a designer's intention and provide a framework for making the structure of a system more explicit. Intermediate design decisions are to be recorded in this language so that there is available a complete trace of the evolution of the design, including alternative designs that were considered and discarded.

e) A methodology for data base system development. This is to be an application of our methodology to data base systems. With such an application in mind we can specify the steps in the step-wise refinement process. This application of our methodology will help to verify the practicality of our approach.

f) A conceptual schema definitional facility. This is to be an aid for software system designers, especially data base designers, to model their understanding of the system application and its environment so that they can formulate clearer and more precise requirements for the system than they otherwise would.

g) Language constructs to support the specification, design and verification of concurrent programs. These are to overcome the problems that arise when most design methodologies are applied to concurrent programs. These constructs are to support high-level abstraction, to support a general communication concept and to allow step-wise refinement to be as easy for concurrent programs as it is for sequential programs.

III RESEARCH ACCOMPLISHMENTS

Our research group has made substantial progress toward development of these aids. We have worked out a software development methodology, applied it to data base system design, developed performance models, specification languages, conceptual schema facilities and communication ports for concurrent programs. Some of these aids were developed as thesis projects by graduate students in our group. The others were developed by or under the direction of faculty. The following sections describe for each research objective what we accomplished and where more details can be found in journal papers and technical reports.

A. Methodology for Software Development

We believe that there are two well-established engineering principles which should be added to software engineering:

a) design evaluation, and

b) development of "mock-up" models as an aid to designers.

We know of only limited work in this area [Chester and Yeh 1977].

It is our conviction that these two principles could well be included in software design processes to complement the current methodology of step-wise refinement [Dijkstra 1972] in software development. We have proposed a software design methodology [Yeh and Baker 1977, Chester and Yeh 1977, Yeh 1978b, and Baker, Chester and Yeh 1978] based on the philosophy of step-wise evaluation and refinement by using three models to characterize the design process as shown in Figure 1.

The system structure model is the usual model for decomposing the system into a hierarchical structure of levels of abstractions [Robinson and Levitt 1977, Dijkstra 1968] using the step-wise refinement approach. The system documentation model is a mechanism for documenting all the intermediate design decisions of the system so that modification during the design process or in the future during system evolution can be accomplished clearly and meaningfully. The hierarchical performance evaluation model is used to evaluate the system design as the design unfolds. It is important to note that during the design process, all

three models interact and should be refined simultaneously.

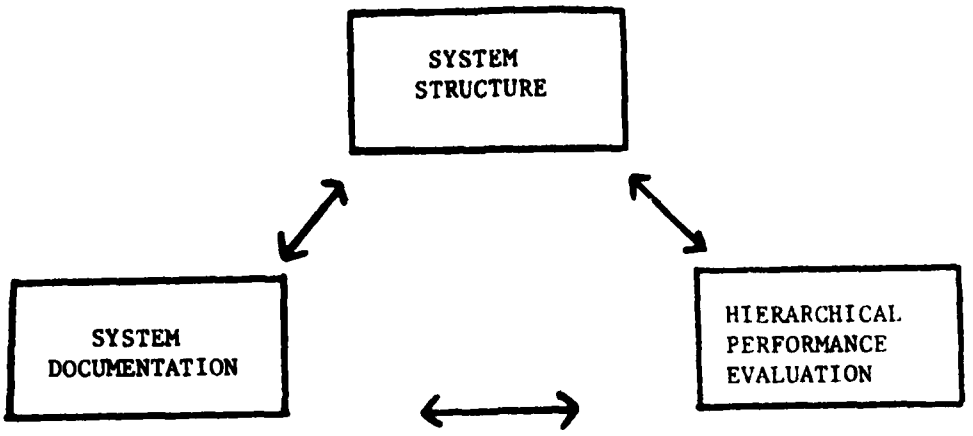


Figure 1 - A Model for Software Design Process.

Aspects of these models are discussed in more detail elsewhere in this report. In particular, The hierarchical performance evaluation models are discussed in section B, the system structure models in C and G, and the system documentation models in D and F.

One of the problems that arise with a new methodology is that a large body of software has already been developed that is incompatible with that methodology. In the case of our methodology, most previously written software is incompatible because it is not organized into a hierarchy of abstract machines as reflected in the software modules. During the last year of research we examined the problem of restructuring existing software so that it can evolve (be maintained) within the framework of our development methodology. We took as our test case the problem of restructuring COBOL programs into programs based on abstract data types (module definitions that implement abstract machines) because there are so many COBOL programs and programmers. If

COBOL programmers can benefit from our development methodology and if they adopt it, it will significantly change a large part of the computing community.

We have implemented an extension to COBOL, called XTC, which allows COBOL programmers to define abstract data types. XTC [Chester 1980] is a small extension; it involves some additions to COBOL syntax that allow programmers to group subroutines with the data structure on which they operate and to declare such data structures without repeating their internal formats. A preprocessor [Hartman 1980a] has been written in COBOL so that XTC programs can be compiled with a standard COBOL compiler.

We have studied the problem of transforming COBOL programs with large global referencing environments into a set of procedural and abstract data type modules. A methodology is presented in [Hartman 1980b] for performing this restructuring in a conservative manner, i.e., so that as few program statements as possible are changed and the restructured program is functionally equivalent to the original program. The steps of the methodology that require design decisions are clearly distinguishable from those that can be automated. The methodology uses a "data-driven decomposition" heuristic that is justifiable for many COBOL data processing programs. Following the setting of goals and initial program understanding, the program's data structures are partitioned to form the kernals of abstract data type modules, each with a concept and an inclusion rule for assigning statements to it. A combination of flow analysis and symbolic execution are used to create the operations of each module, and the calls and parameters which preserve the original logic and data access. The methodology has been demonstrated by restructuring a 1500 line production COBOL program into ten modules. The XTC language was useful for coding the modules.

B. Hierarchical Performance Models

1. Hierarchical Performance evaluation

The success or failure of any system, of course, depends greatly upon the level of performance which it achieves during actual operation. Current approaches to performance evaluation rely heavily on monitoring

a system after its design and implementation and then fine-tuning to achieve a desirable level of performance. In many cases, however, major redesign is required - an effort which may account for a large portion of the system development cost.

This section contains a very brief description of a performance evaluation technique which can be used with the hierarchical design approach and which seems to have several advantages over current performance evaluation procedures. A fuller description is found in [Baker, Chester and Yeh 1978] and in [Baker (Ph.D. thesis) 1978]. This technique involves the construction of a hierarchical performance evaluation model (HPEM). The purpose of this model is two-fold:

a) To provide the designer with feedback at each step of the design process regarding the performance characteristics of the system, and

b) to provide a basis for choosing between alternative designs at each level.

The HPEM is developed in parallel with the system design and is, likewise, hierarchically structured. As the design unfolds new levels are added to the HPEM - each reflecting increased knowledge about the implementation of the system and the resulting effects upon system performance. Thus, the HPEM allows the designer to derive as much information as possible about the potential performance characteristics of the current design. Constant interaction with the HPEM enables the designer to "guide" the system down an appropriate design path with a minimum of redesign and backtracking.

The addition of each level to the HPEM consists of three distinct phases:

a) parameterization,

b) construction of an analytical model, and

c) queuing network simulation.

In the following sections we will briefly describe each process.

2. Parameterization

The parameterization of the ith level of the HPEM is concerned with specifying those aspects of the system design at level i which are important with respect to system performance. The designer has complete freedom in choosing the performance parameters and hence design aspects may be represented explicitly, implicitly, or not at all in the HPEM. The parameterization is modular with respect to the system design. That is, for each data abstraction of system level i a performance parameter specification is developed - this specification reflecting the properties of the data abstraction relevant to system performance.

The performance parameters of level i can be classified as design parameters or implementation parameters. The design parameters, representing aspects of the design which have been established, may be further classified as scenario parameters or control parameters.

Scenario parameters represent aspects of the state of the system which are determined by the user applications. The value of a scenario parameter at level N is determined by an expected workload specification. At level i (0 <= i <= N) the value of each scenario parameter is derived from a scenario parameter mapping function which defines the value of the parameter as a function of the performance parameters of level i+1.

Control parameters represent aspects of the system design which may be varied to enhance the performance of user applications.

During the evaluation process, the system designer may use different values of control parameters in order to determine the most appropriate configuration of the system for an application.

Implementation parameters represent aspects of the system design which have yet to be realized but which must be included in the model for proper evaluation of the system. Implementation parameters at level i, for example, would include the designer's estimates for the execution speeds of level i operations. The values of such parameters represent constraints on the design which have yet to be satisfied.

3. Analytical Model

The analytical model consists of analytical equations which define the cost of a particular design (completed through level i) in terms of time expressions for the operations of level i and estimates of storage requirements.

Using an approach similar to the one used by Wegbriet[1976], it is possible to analyze a program module at level i in order to derive a cost expression in terms of the model performance parameters at level i+1.

The designer can then evaluate the cost of executing this module for various values of the model parameters.

Performing such analysis on all modules at each design step thus enables the designer to construct a cost expression for the design in terms of the parameters of the most recently completed level.

4. Simulation Model

Unfortunately, many aspects of complex systems cannot be captured by an analytical model. In data base systems, for example, performance can be significantly affected by such things as contention for resources caused by multiple users, locking protocol, and creation of temporary data objects - factors which cannot easily be incorporated into an analytical model. Likewise, a simple time expression may not adequately characterize complex, parameterized operations. Thus, we propose that the third phase of HPEM be a simulation of the operation of the system design.

The basis for the simulation process is a queuing network model constructed by the designer. It differs, however, from the typical queuing models in that it attempts to model contention for abstract resources (relations, files, directories, indexes, etc.) defined in the software rather than hardware resources such as CPUs, disks, and I/O devices. The structure of the model reflects that of the system itself, that is, a hierarchy of queuing networks is constructed each of which models the performance characteristics of the system design at a particular level of abstraction. Thus, successive networks in the

hierarchy represent increasingly detailed models of the system performance. Moreover, adjacent networks are "connected" in that the solution of each network can be used to parameterize the immediately preceding one.

C. Specification Language

We developed two specification languages. The first one, reported in [Chester and Yeh 1977] and [Baker, Chester and Yeh 1978], was a language for specifying abstract machines in terms of their internal states, represented by list structures. The specification of a machine consisted of a description of its initial state, the effects of various operations on the machine, the exception conditions under which the operations are not applicable, and, if the operations are functions, the values they return. With the help of an interpreter for this specification language, we could "run" the specifications as programs, providing us with a "mock-up" of the system to be designed. By experimenting with the running specifications we could see whether they behaved as intended, thereby allowing us to catch errors in the specification before design work has begun.

One problem with this specification language, besides being a very high level programming language, was that it encouraged the use of "hidden" functions: functions that are defined to simplify (or, sometimes, to make possible) the definition of operations, but which themselves were not going to be implemented in the final system. Dissatisfaction with this aspect of the language led to the development of the second specification language, which is described in [Chester 1979]. This specification language is based on the concept of traces, which are sequences of operator calls on an abstract machine. By identifying the states of abstract machines with the sequences of operator calls that produce them and by relating these states to the values that subsequent function calls will return, a designer can specify the behavior of abstract machines purely in terms of their "external" properties. No speculation or commitment to the internal structure of the machines is required.

D. Design Documentation Language

Although we have not implemented a design documentation language, we have described such a language in much detail in [Chester and Yeh 1977] and [Baker, Chester and Yeh 1978]. This language, which is to be an integral part of the design process and not merely a specification tool that is used "after the fact", contains the following three features for describing global properties of system designs.

1. Specification of Alternative Designs

Consider the implementation of level i in the hierarchical design process. There may exist a large number of modules which may be combined in various ways to produce alternative designs for level i-1. The designer may document in the language a) modules that must appear together in designs, b) modules that represent alternatives to other modules, and c) modules that are optional in the design.

2. Specification of Modules Interconnections

In specifying the interconnections between modules of a system design we are concerned with several different relationships. These include the has access relationship, where one module has access to instances of another module by virtue of the fact that it created that instance or has access to another module that has access to it, and the uses relationship, which indicates the means by which one module may use instances of another module to which it has access. These uses include reading from and writing to modules and creating new instances of modules.

3. Specification of Level Structure

In describing the structure of a particular level design it is desirable to consider specifying the hierarchical relationships (if any) between modules of the level, and partitioning the level into subsystems which may be partially ordered within the level.

E. Methodology for Data Base System Development

Data Base Systems (DBS) were chosen for an initial study and verification of our methodology for the following reasons:

a) DBS will become a necessary tool for almost every institution of even moderate size. They will become some of the most widely used software systems. Yet, surprisingly, no systematic approach exists for their development.

b) We believe that the important problem of designing DBS lends itself naturally to software engineering methodologies. Recently, work by researchers in the data base area [Smith and Smith 1977a, Senko 1968] pointed out that notions such as abstract data types and hierarchical organization are important tools for data base design.

c) DBS can provide a focus for our methodology and tools to gain necessary experience before generalizations can be made. For example, one of the problems in using the top-down approach in design is to determine where the top is. In data base areas, it is generally agreed that the "conceptual schema" should be the top level. Furthermore, there seems to be some agreement on the number of levels in DB design so that a hierarchical methodology can be evaluated against some of the existing systems. On the other hand, DBS can be very sophisticated so that every aspect of the enhanced methodology can be exercised.

d) We believe that the problem of data base design can be approached effectively utilizing the same methodology of hierarchical structures provided a hierarchically structured data base management system exists with a transparent design and clear documentation (e.g., designed with the proposed methodology). This is so because the designer of data bases can take advantage of the level structure of the data base management system and proceed downward from the conceptual schema using evaluation tools.

e) Data bases are part of a variety of development tools, such as the library of documents prepared in the design documentation language, and of many software systems in general.

Our multi-level approach to data base system design is illustrated by Figure 2, which shows how the levels of the DB system design correspond to the levels of our software development methodology. The first step in the design is to obtain a conceptual schema of the information structure; then the conceptual schema is mapped into a

logical schema. The details of physical storage structures are designed in four more levels of mapping with each level introducing more implementation decisions.

Details of our methodology for the design of data base systems are given in [Baker, Chester and Yeh 1978, Yeh, Araya and Chang 1979, Yeh, Chang and Mohan 1978, Baker and Yeh 1977, Yeh and Baker 1977, and Yeh 1978a]. Related information can be found in [Yeh, Roussopoulos and Chang 1978, Mohan 1979a, Mohan 1979b, Mohan 1979c, and Mohan and Yeh 1979].

Our methodology was used on a small scale by Jou and Shih in a data base system for storing chinese characters. The system is partially described in [Shih (M.S. report) 1980].

F. Conceptual Schema Definition Facility

A data base is a model of a portion of the real world. The first steps in data base design are to capture the real-world concepts and to map them into some formal descriptions. Such formal descriptions, or "Conceptual Schemata" (CS) [ANSI/SPARC 1976], should be independent of the DBS used or the actual implementation of the data base. In contrast to a "conceptual schema", a "logical data base structure" refers to the data base structure constrained by the capability of a given DBS, and a "physical data base structure" refers to the details of the storage structures of a data base.

We have developed and implemented a Conceptual Schema Definition Language (CSDL) to ease the translation of unstructured knowledge about the data in a data base system and the uses of that data in an application into a formal conceptual schema. CSDL is a design tool with which the designer can express his understanding of the application. CSDL is described in great detail in [Yeh, Roussopoulos and Chang 1978, and Jou (M.A. thesis) 1979].

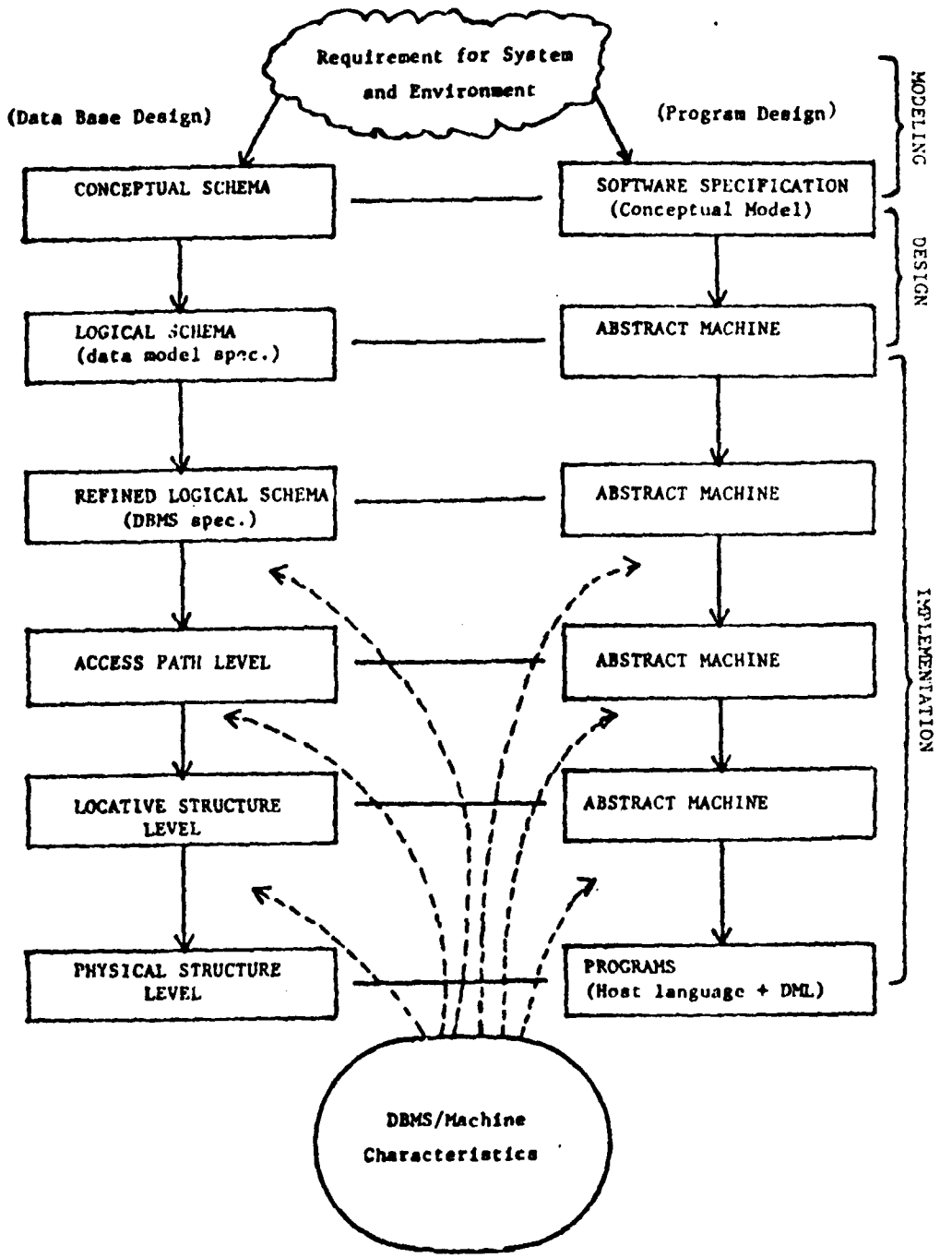


Figure 2 - Multi-level Data Base System Design.

There are two data types in CSDL, concepts and frames. The operators defined on them are divided into three classes, naming, selection and linking. All three classes of operators appear in every data definition facility. The first class allows the designer to name objects (concepts), and relationships among them. The second corresponds to specialization or particularization, namely the means for deriving from a general concept a more specialized one. Linking is the grouping together of two or more concepts or relationships to form another meaningful relationship. Linking is very general and is used to express relationships among concepts and/or frames that were formed by other linking operations.

The operators are the only direct means for creating and manipulating the abstract objects that model the application. By adhering to a strict programming discipline, we can guarantee the syntactic correctness of the created conceptual schemata. In addition, the high level operators of CSDL free the designer from having to know the internal representation of concepts and frames and the task of making it consistent.

CSDL has relevance to software development methodology as well as data base design because it or something like it may be useful for formalizing the requirement specifications of software systems. This is so because software systems can be viewed as "models of the real world". If, for instance, we consider an information system for a hospital, or a real-time system that controls chemical processes, we see that those systems reflect to a large extent the environment that they have to control and the policies or goals of the organization to which they belong. From this perspective it is important to reach a high level understanding of the environment and the set of goals or policies that constitute the context of the software system, and this would be greatly facilitated by a tool like CSDL. Once that understanding has been modelled, it will be much easier to extract from it the exact requirements of the software system under development.

G. Concurrent Programming Constructs

Although many design methodologies exist, they are not all adequate for use in the environment of concurrent programs. In order to incorporate parallelism into these methodologies, we need concurrent programming constructs that decompose problems into manageable parts and are helpful in exploring the parallelism that is possible in the system being designed. We have introduced three such constructs: communication ports, behavior controllers and structural locks. These mechanisms constrain concurrent programs so that undesirable effects such as deadlock cannot occur while at the same time they leave the designer with considerable freedom to take advantage of concurrency in his design.

A communication port (CP) is a general mechanism to specify the interaction among software components which can proceed in parallel. It is similar to Brinch Hansen's distributed process [Brinch Hansen 1978] with two additional capabilities: CP specifies precisely which processes can communicate with one another as well as when the communication between two processes should be disconnected. The second capability is derived from the notion of "disconnect time" which can be used to model various communication mechanisms such as read/write and the conventional procedure call/return. This capability allows the programmer to specify a high degree of overlapping of processes and thus is a tool to develop efficient concurrent programs. The first capability provides support for abstraction and data protection in the same fashion as in other high level (sequential) programming languages. Thus, the use of this mechanism makes the resulting programs easier to comprehend and more reliable.

Every communication port has its own name. Each port has one master process and one or more servant processes. Master and servants can communicate with each other via CP. The difference between master and servant is that the master has the right to disconnect the communication at the time it chooses once the communication is set up, while the servant has no such right. For more details see [Mao and Yeh 1979a, Mao and Yeh 1979b, Mao and Yeh 1980, Mao (M.A. thesis) 1979 and Mao 1980].

Behavior controllers are based on the observation that in a properly structured computing environment, the external behavior of a process can be restricted to the performance of externally identified operations on data objects. By properly constraining and scheduling a process's external behavior one can deal with many of the problems of concurrency. There are two kinds of behavior controllers: 1) rights controllers, that enforce constraints on the sequence of operations performed by a process through a particular data object access path, and 2) synchronizing controllers, that schedule the execution of operations on a shared data object to achieve specified sequence constraints which are independent of the access path (and, therefore, process) through which the operations are performed.

Abstractly, both kinds of behavior controllers are finite state machines which change state with each operation on the data objects. Thus both a process and the data object it wants to manipulate have states associated with them, and the process can perform an operation on the data object only if there is a transition from the process state and a transition from the data object state which correspond to that operation. For more details, see [Conner (Ph.D. thesis) 1979 and Conner 1979].

For concurrency in data base systems we have devised structural locks. These mechanisms lock certain fields of certain records from other processes. There are simple tests, based on set operations applied to sets of record identifiers and sets of attributes appearing in DBS transactions, that effectively determine when conflicts will occur so that transactions can be scheduled for maximum parallelism. It is expected that structural locks will be especially effective on machines having associative architectures. For more information see [Lee and Yeh 1979a, Lee and Yeh 1979b and Lee (Ph.D. thesis) 1979].

We also did some work of a more theoretical nature. An algorithm for determining when a computation involving parallel processes has terminated, due to Dijkstra and Scholten, was generalized in [Chandy and Misra 1979]. A simple model of concurrent programs was made by conservatively extending known ideas in sequential programming. This model is reported in [Chandy and Misra 1980].

PROFESSIONAL PERSONNEL ASSOCIATED WITH THIS PROJECT

K. Mani Chandy

Philip Chang

Daniel Chester

Jayadev Misra

Nick Roussopoulos

Raymond T. Yeh

STUDENTS RECEIVING DEGREES WHO WERE ASSOCIATED WITH THIS PROJECT

Jerry Baker, Ph.D. in computer science, 1978.
"Hierarchical Performance Evaluation"

Michael Haden Conner, Ph.D. in computer science, 1979.
"Process Synchronization by Behavior Controllers"

Emery Dze-Min Jou, M.A. in computer science, 1979.
"A Conceptual Schema Definition Language
Design and Implementation"

Sukho Lee, Ph.D. in computer science, 1979.
"Locking Mechanisms for Concurrency Control and
Consistency in Data Base Systems"

Tsang William Mao, M.A. in computer science, 1979.
"A Study on the Language Features for
Concurrent Programming"

Lory Pei-Yu Shih, M.S. in computer science, 1980.
"A Portable System for Chinese Character Input
and Output Processing"

CHRONOLOGICAL LIST OF PUBLISHED PAPERS

Chandy, K. M., 1977, A Data Base System for Generating Linear Programming Models, ORSA/TIMS Conf., Los Angeles.

Chester, D., and Yeh, R. T., 1977, Software Development by Evaluation of System Design, Proc. COMPSAC77, Chicago, Illinois, Nov., 431-441.

Yeh, R. T. and Baker, J., 1977, Toward a Design Methodology for DBMS: A Software Engineering Approach, Proc. 3rd VLDB Conf., Tokyo, Japan.

Yeh, R. T., Roussopoulos, N., and Chang, P., 1978, Data Base Design - An Approach and Some Issues, Data Base Technology, INFOTECH State of the Art Report.

Baker, J., Chester, D., and Yeh, R. T., 1978, Software Development by Stepwise Evaluation and Refinement, Software Revolution, INFOTECH State of the art report.

Yeh, R. T., 1978a, Data Base and Software Engineering - a Design Viewpoint, Proc. 4th VLDB, West Germany.

Yeh, R. T., Chang, P., and Mohan, C., 1978, A Multi-Level Approach to Data Base Design, Proc. COMPSAC78, Chicago, Illinois.

Chandy, K. M. and Misra, J., 1979, A Generalization of Termination Detection for Diffusing Computations, submitted to CACM.

Chester, Daniel, 1979, An Approach to Abstract Specification Based on Traces, Proc. COMPSAC79, Chicago, Illinois, Nov., 123-127.

Lee, S. K. and Yeh, R. T., 1979a, Concurrency Control in Distributed Environment, Distributed Data Bases, INFOTECH State of the Art Report.

Lee, Sukho and Yeh, Raymond T., 1979b, Structural Locking for Concurrency Control in Data Base Systems, Proc. COMPSAC79, Chicago, Illinois, Nov., 123-127.

Mao, T. W. and Yeh, R. T., 1979a, Communication Port: A Language

Concept for Concurrent Programming, Proc. of Distributed Computing Systems Conf., Oct.

Mao, T. W. and Yeh, R. T., 1979b, An Illustration of Systematic Design of Parallel Programs for Real-time Applications, Proc. COMPSAC79, Nov., 392-397.

Mohan, C. and Yeh, R. T., 1979, Distributed Data Base Systems - A Framework for Data Base Design, Distributed Data Bases, INFOTECH State of the Art Report, Infotech International.

Yeh, R. T., 1979, Notes on Programming Methodology, Encyclopedia of Computer Science and Engineering.

Chandy, K. M. and Misra, J., 1980, A Simple Model of Distributed Programs Based on Implementation-Hiding and Process Autonomy, ACM SIGPLAN Notices, Vol. 15, No. 7, July.

Mao, T. W. and Yeh, R. T., 1980, Communication Port: A Language Concept for Concurrent Programming, IEEE Transactions on Software Engineering, SE-6, 2, March, 194-204.

TECHNICAL REPORTS AND OTHER PAPERS

Baker, J. and Yeh, R.T., 1977, A Hierarchical Design Methodology for Data Base Systems, TR-70, Department of Computer Sciences, University of Texas at Austin, Austin, Texas.

Yeh, R. T., 1978b, Notes on Programming Methodology, Technical Report SDBEG-1, Department of Computer Sciences, University of Texas at Austin, Austin, Texas.

Baker, J., Chester, D. and Yeh, R. T., 1978, Software Development by Stepwise Evaluation and Refinement, Technical Report SDBEG-2, University of Texas at Austin.

Yeh, R. T., Roussopoulos, N. and Chang, P., 1978, Data Base Design - An Approach and Some Issues, Technical Report SDBEG-4, Department of Computer Sciences, University of Texas at Austin, Austin, Texas.

Mohan, C., 1978, An Overview of Recent Data Base Research, Technical Report SDBEG-5, Department of Computer Sciences, University of Texas at Austin, Austin, Texas.

Chandy, K. M. and Misra, J., 1979, A Generalization of Termination Detection for Diffusing Computations, Technical Report TR-106, Department of Computer Sciences, University of Texas at Austin, Austin, Texas.

Mohan, C., 1979a, Some Notes on Multi-Level Data Base Design, Technical Report TR-128, Department of Computer Sciences, University of Texas at Austin, Austin, Texas.

Mohan, C., 1979b, Distributed Data Base Management: Some Thoughts and Analyses, Technical Report TR-129, Department of Computer Sciences, University of Texas at Austin, Austin, Texas.

Mohan, C., 1979c, Data Base Design in the Distributed Environment, Technical Report TR-131, Department of Computer Sciences, University of Texas at Austin, Austin, Texas.

Yeh, Raymond T., Araya, Agustin and Chang, Philip, 1979, Software and Data Base Engineering - Towards a Common Design Methodology,

Technical Report SDBEG-6, Department of Computer Sciences, University of Texas at Austin, Austin, Texas.

Mohan, C. and Yeh, R. T., 1979, Distributed Data Base Systems - A Framework for Data Base Design, Technical Report SDBEG-10, Department of Computer Sciences, University of Texas at Austin, Austin, Texas.

Mao, T. W. and Yeh, R. T., 1979a, Communication Port: A Language Concept for Concurrent Programming, Technical Report SDBEG-12, Department of Computer Sciences, University of Texas at Austin, Austin, Texas.

Mao, T. W. and Yeh, R. T., 1979b, An Illustration of Systematic Design of Parallel Programs, Technical Report SDBEG-13, Department of Computer Sciences, University of Texas at Austin, Austin, Texas.

Lee, Sukho and Yeh, Raymond T., 1979b, Structural Locking for Concurrency Control in Data Base Systems, Technical Report SDBEG-14, Department of Computer Sciences, University of Texas at Austin, Austin, Texas.

Conner, Michael Haden, 1979, Process Synchronization by Behavior Controllers, Technical Report SDBEG-16, Department of Computer Sciences, University of Texas at Austin, Austin, Texas.

Chandy, K. M. and Misra, J., 1980, A Simple Model of Distributed Programs Based on Implementation-Hiding and Process Autonomy, Technical Report TR-127, Department of Computer Sciences, University of Texas at Austin, Austin, Texas.

Chester, Daniel L., 1980, An Abstract Data Type Extension to COBOL, Technical Report SDBEG-17, Department of Computer Sciences, University of Texas at Austin, Austin, Texas.

Hartman, John, 1980a, XTC User's Guide, Technical Report SDBEG-18, Department of Computer Sciences, University of Texas at Austin, Austin, Texas.

Hartman, John, 1980b, A Methodology for Restructuring COBOL Programs into Abstract Data Type Modules, *Technical Report (in preparation)*,

Department of Computer Sciences, University of Texas at Austin, Austin, Texas.

Mao, Tsang William, 1980, A Study on the Language Features for Concurrent Programming, Technical Report SDBEG-19, Department of Computer Sciences, University of Texas at Austin, Austin, Texas.

REFERENCES

(Not Supported by Grant)

ANSI/X3/SPARC study group on Data Base Systems, 1976, Interim Report, ACM-SIGMOD Newsletter, FDT 7, 2.

Brinch Hansen, P., 1978, Distributed Processes: A Concurrent Programming Concept, CACM, Nov.

Dijkstra, E., 1968, The Structure of T.H.E. Multiprogramming System, CACM, 11, 5, 341-346.

Dijkstra, E., 1972, Notes on Structured Programming, in Structured Programming, (ed. Hoare), 1-82, Academic Press.

Robinson, R. and Levitt, K., 1977, Proof Techniques for Hierarchically Structured Programs, Current Trends in Programming Methodology - Vol 2 - Program Validation, (ed. Yeh), Prentice-Hall, inc.

Senko, M. E. etc., 1968, A File Organization Evaluation Modem (FOREM), Information Processing 68.

Smith, J. M. and Smith, D. C. P., 1977a, Data Base Abstraction, CACM, 10, 6.

Wegbreit, B., 1976, Verifying Program Performance, JACM, 23, 4, 691-699.